Globalizing Selectively: Shared-memory Efficiency with Address-space Separation

Nilesh Mahajan
Indiana University
Bloomington, IN 47405
nnmahaja@cs.indiana.edu

Uday Pitambare
Indiana University
Bloomington, IN 47405
upitamba@cs.indiana.edu

Arun Chauhan
Indiana University
Bloomington, IN 47405
achauhan@cs.indiana.edu

ABSTRACT

It has become common for MPI-based applications to run on shared-memory machines. However, MPI semantics do not allow leveraging shared memory fully for communication between processes from within the MPI library. This paper presents an approach that combines compiler transformations with a specialized runtime system to achieve zero-copy communication whenever possible by proving certain properties statically and globalizing data selectively by altering the allocation and deallocation of communication buffers. The runtime system provides dynamic optimization, when such proofs are not possible statically, by copying data only when there are write-write or read-write conflicts. We implemented a prototype compiler, using ROSE, and evaluated it on several benchmarks. Our system produces code that performs better than MPI in most cases and no worse than MPI, tuned for shared memory, in all cases.

1. INTRODUCTION

Message Passing Interface (MPI) is well established as the programming interface of choice for writing scalable parallel programs. Based on communicating sequential processes (CSP) [9], MPI assumes a partitioned address space. Arguably, even though programming with partitioned address spaces might, at first, appear to be more difficult than shared-memory programming, it has certain advantages over the latter, primarily stemming from the fact that there are no shared data that might get corrupted from hidden race conditions. Another, performance-related advantage is the inherently better data locality afforded by partitioned address space programming. Therefore, a case could be made for using CSP-style programming even when the system provides shared-memory hardware, while the compiler and the run-time system work together to leverage the shared memory in order to minimize communication costs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503275

Additionally, as the number of cores on a chip grows, it becomes possible to run bigger parallel programs on a single shared-memory *node* that might have run only on a multi-node (clustered) parallel machine before. As a result, it would be desirable to be able to use the many-core shared-memory machines to run those programs that were written for clusters, using MPI, while making the best use of the available shared memory.

Finally, programming on large clusters increasingly involves hybrid programming models involving both message passing and shared data. Such hybrid programming strategies are based on the premise that message-passing programming using MPI is not able to make the best use of hardware shared memory. Unfortunately, this exacerbates the already difficult problem of writing, debugging, and maintaining parallel programs. If intra-node performance could be improved within a single programming model, such as one based on MPI, a large number of programs could avoid having to resort to the complex and error-prone hybrid parallel programming.

Motivated by these observations, we have devised a compiler strategy to enable MPI programs to make better use of hardware shared memory. Our strategy relies on two components: 1. A source-level compiler that transforms a program using MPI into one that uses our run-time system, with some additional safeguards for correctness; and 2. A runtime system that is a replacement for MPI primitives. The transformed code uses independent processes, as before, but utilizes a block of shared memory for allocating communication buffers. Combined with some additional meta-data to keep track of the buffer usage, this makes it possible to communicate without having to copy data from the sender to the receiver. In cases where the compiler is unable to reason about communication buffers statically the run-time system is able to avoid copying data, whenever possible, thus providing graceful degradation of performance. Our design automatically achieves *ownership transfer*, whenever possible, which has been proposed as an optimization technique for shared-memory transport [13, 6].

The run-time system uses the meta-data associated with each communication buffer to implement a multiple-readers single-writer method for sharing the buffers. We use a *copy-on-contention* scheme to copy the buffer, if needed, instead of serializing access when there are multiple writers. In this way, our strategy implements a logical single-assignment approach for communication data, while sharing read-only data as much as possible.

We have implemented our compiler transformations using

the ROSE source-level compiler framework for C++ [14] and evaluated our strategy on a series of micro-benchmarks. We also evaluated its performance on six diverse benchmarks. Communication performance ranges from 1x to 10000x compared to MPI, depending on data size and the exact communication primitive, and application kernel performance increases up to 10x. Our prototype library can meet the performance of highly tuned MPI even in the worst-case.

We make the following contributions:

- We identified and applied a set of compiler transformations that are applicable to a large majority of MPI programs, in order to avoid copying of data when hardware shared memory is available.
- We designed a smart run-time system that provides a communication library to replace all the standard MPI primitives in order to fully leverage the shared memory. We use the idea of *copy-on-conflict* to eliminate delays that would otherwise be caused due to sharing of buffers.
- We implemented a proof-of-concept compiler for C++, using the ROSE source-level transformer, and an accompanying run-time library, and evaluated the performance of our system on several benchmarks and micro-benchmarks.

2. BACKGROUND

Optimizing MPI for shared memory has been recognized as an important issue and several MPI libraries implement versions of the MPI primitives specially optimized for shared memory. However, any implementation of MPI must preserve the semantics of the primitives. This requirement constrains the types of optimizations that the library might be able to perform. For example, consider the following piece of C-like pseudocode that uses MPI to send data.

```
produce(buffer);
if (sender) {
  MPI_send(buffer, destination); produce(buffer);
} else { MPI_receive(buffer, source); }
```

The function `produce` fills in `buffer`, which is then sent to `destination`. The sender then immediately proceeds to rewrite `buffer` with new values. As soon as `MPI_send` finishes, the program is free to deallocate or reuse the send buffer. Implementing zero-copy communication would require:

1. Allocating buffer in shared memory.
2. Pointing the receiver's buffer to the sender's buffer.
3. Ensuring that the sender does not overwrite the data that the receiver has not finished consuming.

It is infeasible for a library to achieve all these, since it cannot change buffer allocations (except if all buffers are on the heap and the library implements its own heap allocator), it cannot change the buffer pointers without changing MPI functions signatures (which would break the API), and cannot ensure correctness when multiple writers are present without using operating system services to implement something like page-level copy-on-write.

Consider how a library might attempt to implement each of these steps. Allocating buffers in shared memory would require the library to implement its own heap allocator such that all heap allocation happens in shared memory. This would complicate the library. Even with its own heap allocator, it would not be possible to share statically allocated or stack-allocated buffers.

Pointing the receiver's buffer to sender's buffer would involve changing the signature of `MPI_receive` so that the buffer argument passed to it could be modified. However, the signature must be changed in such a way that the existing MPI code remains unchanged, which could be done, for example, by using C++ references. However, changing the signature makes the library non-compliant with the MPI standard [12].

Finally, there is no easy way for a library to tell when a receiver is done consuming the data it received. If the receiver attempts to free its receive buffer that would be one indication that it no longer needs the data. In such a case, the library could delay the return from `MPI_send` until the receiver attempts to free the receive buffer. However, the receiver may never free the buffer, or free it after an arbitrary amount of delay. In either case, it would not be practical to delay the sender indefinitely. Further, the sender might not overwrite its buffer, in which case delaying the return of `MPI_send` would be counterproductive.

Given these constraints, it is reasonable to conclude that a purely library-based strategy would not be able to implement zero-copy communication on shared memory for MPI programs.

One alternative is to add support in the operating system to mark certain shared pages read-only, with copy-on-write policy. The MPI library could then mark a communication buffer read-only, which would be copied if a sharing process attempted to write into it. However, even a single writer might trigger copying, defeating the original goal of avoiding data copying. If the process was going to completely overwrite the original content the library would still need to copy the original content, since it does not know the program behavior in advance, only to be overwritten immediately. Finally, such an approach would require kernel modifications and introduce additional system call overheads.

One option is be to convert the MPI program into a multi-threaded one, as against our approach of using shared memory to hold selected data. This is the approach taken by Friedley et al., which works by extending the MPI primitives with ownership-transfer calls [6]. Similar to Shen et al. [16], they also provide compiler analysis to translate MPI programs into multi-threaded version. We chose not to do that for five reasons:

- Converting processes to threads drastically changes the program semantics related to sharing of statically allocated data. Such an approach may even be infeasible if the original code involves closed-source libraries and / or thread-unsafe libraries.
- When feasible, it involves making changes to large portions of the original code, including in code that may not be directly related to MPI or communication.
- We would still have to deal with synchronization issues to orchestrate accesses to shared buffers.
- On Linux, which is a widely used platform for HPC, there is little performance benefit of using kernel-level threads compared to using processes due to the way Linux implements threads [15].
- If we were to extend our approach to handle multi-threaded MPI programs, converting processes to threads is likely to lead to further complications.
- An approach that works with partitioned address spaces (i.e., processes) will also be suitable for other scalable programming models that use partitioned address spaces, such as Partitioned Global Address Space (PGAS) models.

3. OVERALL APPROACH

We transform Single Program Multiple Data (SPMD) MPI programs to *globalize* communication buffers selectively by allocating them in shared-memory and using Single Writer Multiple Readers strategy to orchestrate shared access.

Figure 1 shows a simple example to illustrate the translation. We use C++-style syntax for all the examples. Functions in the `MPI` namespace are MPI functions and those in the `shm` namespace are the shared-memory equivalent from our library. The pseudocode code on the left encodes a producer sending data to a consumer. The code on the right performs that same computation using shared memory, changed as follows:

- Allocation and deallocation of buffers are changed to shared memory, using `shm:allocate` and `shm::release`.
- Calls to `MPI::Send` and `MPI::Receive` are replaced by calls to `shm::send` and `shm::receive`.

`shm::send` simply records the buffer pointer in the receiver's queue. `shm::receive` copies the buffer pointer it finds in the receive queue to its buffer argument, which is passed by reference. If the queue is empty `shm::receive` blocks until a corresponding `send` is posted. Note that we have omitted other arguments, such as communicators and tags for the sake of clarity, even though they are supported by our library.

The strategy as described would work as long as the communication buffers are not reused. However, it is common for communication buffers to be reused. For instance, the producer in Figure 1 repeatedly generates new data in the same buffer. The problem is caused by aliasing of sender and receiver buffers that did not exist in the original code since they were in distinct address spaces. As a result, the producer must make sure that the buffer is no longer being consumed before it overwrites the data in it. If the buffer is still in use the producer could wait for the data to be consumed fully. Instead, we implement a *copy-on-conflict* strategy [10]. If the producer detects that the buffer is still in use by the consumer it allocates a new buffer for itself.

The copy-on-conflict strategy is implemented using a readers count(ref-count) for each buffer, which is part of the meta-data associated with each buffer. When a buffer is allocated by the producer using `shm::allocate`, its ref-count is set to 1. The compiler replaces the buffer deallocation by `shm::release`, as Figure 1 shows, which decrements the ref-count and frees the buffer if the count reaches 0. When the buffer is received, its ref-count is incremented by `shm::receive`. Thus, both producer and consumer can continue to share the buffer for reading. A write to the buffer is preceded by a call to `shm::check_clone` that checks the buffer ref-count, and if the count exceeds one it clones the buffer and updates the buffer pointer to point to the clone.

In addition to explicit releases, a buffer's ref-count may also be decremented by `shm::receive`, `shm::send`, and `shm::check_clone`. Both `send` and `receive` decrement the count of the original buffer before updating the buffer pointer to point to the sender's, since it is no longer going to be accessible on the receiver[1]. `shm::check_clone` decrements the ref-count for the original buffer if it was greater than one, and a new allocation is necessary. Notice that after a new buffer has been allocated for the writer and if the ref-count for the original buffer drops to one, the other sharer of the buffer could now write into it without needing any further allocations.

As an optimization, if the compiler can prove that the buffer is going to be completely overwritten, it simply allocates a new buffer, without needing to clone the original. This is accomplished by using `shm::check_alloc`, instead of `shm::check_clone`.

It is important to emphasize that it does not matter if the original sender or the receiver of the buffer was about to write into the buffer—any potential writer would allocate a new buffer or clone the existing one if more than one reader were currently accessing the buffer (the count includes the writer process). Also, there is no explicit manipulation of the ref-count in the program.

The compiler needs to make several changes in the code for the above strategy to work. These fall into three categories:

1. Changing allocation and deallocation of communication buffers, possibly inter-procedurally, so that they live in shared memory.
2. Inserting calls to `shm::check_clone` before a buffer is about to be overwritten after participating in a communication step.
3. Replacing calls to MPI library by their `shm` equivalents.

If the compiler cannot safely convert a buffer's allocation to shared memory, it falls back to using `shm::send_copy` (or `shm::receive_copy`). The `copy` variants behave identically to the default MPI primitives, thus guaranteeing correctness. In other words, it is safe to always replace a primitive `MPI::p` by `shm::p_copy`.

4. PRELIMINARIES

We begin with a series of observations that guide our compiler transformations.

OBSERVATION 1. *For any MPI primitive `p`, it is always valid to replace `MPI::p` by `shm::p_copy`.*

PROOF. The proof of this statement relies on the design of the runtime system, as discussed in Section 5. □

OBSERVATION 2. *It is always valid to allocate communication buffer in shared memory.*

PROOF. The correctness follows from the fact that each process continues to operate in its own address space, so that any allocated buffers do not automatically become visible to other processes. □

This allows the compiler to safely move the allocation of any communication buffer to shared memory. That reduces two copies to one for any communication step, since the data may be moved directly from sender buffer to receiver buffer, provided the receive is posted before the send. However, not all allocations of a variable can be changed, e.g., if the variable is allocated through a library or a separately compiled function.

Definition: *Unknown allocation and deallocation* If a pointer is assigned to from an unknown function or its reference or address passed to an unknown function, the pointer has an unknown allocation. If a pointer is passed to an unknown function and it is not live after that call then the pointer has an unknown deallocation.

The above definition is, clearly, conservative. In general, it is possible (although, not very common) for a single variable

[1]Handling of aliasing is described in Section 7.

```
1  Data *buffer;
2  buffer = new ...;
3  while (1) {
4    if (sender) {
5      produce_data(buffer);
6      MPI::Send(buffer, destination);
7    } else {
8      MPI::Receive(buffer, source);
9      consume_data(buffer);
10   }
11 }
12 delete buffer;
```

⇒

```
1  Data *buffer;
2  buffer = shm::allocate(...);
3  while (1) {
4    if (sender) {
5      shm::check_clone(buffer);
6      produce_data(buffer);
7      shm::send(buffer, destination);
8    } else {
9      shm::receive(buffer, source);
10     consume_data(buffer);
11   }
12 }
13 shm::release(buffer);
```

Figure 1: A simple example illustrating the basic optimization of MPI programs for shared memory. Buffer reuse on the producer may corrupt the data on consumer due to dependencies that did not exist in the original partitioned address-space MPI code. This necessitates a check (`shm::check_clone`) to verify that there is only one writer of the buffer, and clone the buffer, if necessary.

to be allocated (and deallocated) in different ways based on runtime conditions. Therefore, we can talk about multiple allocations of a variable.

Definition: *Shm-allocatable* A variable v is said to be shm-allocatable if it has no unknown allocations or deallocations.

Definition: *Reaching allocation and reachable deallocations* An allocation of buffer b is said to reach a statement s if there is a control-flow path from the allocation to s that does not go through another allocation or deallocation of b. A deallocation of b is reachable from s if there is a control-flow path from s to the deallocation that does not go through another allocation or deallocation of b.

The next two theorems, building on an extended definition of dominance, identify the precise conditions under which send and receive primitives may be replaced by their no-copy version.

Definition: *Dominance by a set* A statement T inside function F is set-dominated by a set of statements S if and only if every control-flow path from the beginning of F to T contains some statement from S.

We can analogously define *post-dominance by a set*. Now, we are ready to state the theorems about introducing no-copy versions.

THEOREM 1. *A send primitive s involving a communication buffer b may be replaced by a no-copy version if: (1) s is set-dominated by a set of known allocations of b reaching s; (2) s is post-set-dominated by a set of known deallocations of b reached from s; and (3) each write of b following s is preceded by a call to* `check_clone`.

PROOF. Since the set of allocations reaching s dominate it and are all known, b can be allocated in shared memory and is guaranteed to be found in shared memory when s executes. Similarly, after s finishes execution, b is guaranteed to be freed due to the post-dominance condition. Note that if the post-dominance condition is violated the shared memory allocation might introduce memory leaks. The call to `check_clone` ensures that data dependencies cause by potential sharing of b are not violated. □

It is not always desirable to share all send buffers guaranteed by the above theorem as Section 7.3 explains.

THEOREM 2. *A receive primitive r involving a communication buffer b may be replaced by a no-copy version if: (1) the size of the receive buffer is identical to the amount of data received into it; (2) each writing of b following r is preceded by a call to* `check_clone`*; and (3) the receive buffer pointer passed to the primitive has a known deallocation.*

PROOF. Since no-copy receive involves altering the passed pointer to point to another buffer in shared memory, the receive operation must overwrite the entire buffer and the program must not access the buffer past its size. As before, cloning the buffer before writing into it in the case of multiple sharers ensures that new data dependencies are not created that did not exist in the original program. Finally, since the original buffer passed to the primitive must be deallocated, it must have a known deallocation. □

Theorems 1 and 2 together form the core of our strategy that we elaborate in the remainder of the paper.

5. RUN-TIME SYSTEM

A core component of our approach is the run-time system that replaces the standard MPI primitives, including point-to-point and collective functions. Note that it is not possible for our library to be a drop-in replacement for MPI since the signatures of some of the equivalent shared-memory functions are slightly different. Moreover, for correctness, the library must be used in conjunction with additional functions to protect against write dependencies that arise (when none

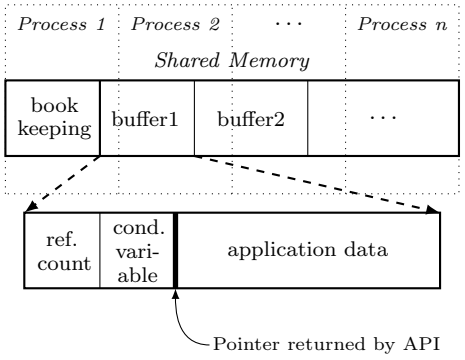

Figure 2: Overview of the run-time system.

```
1  Algorithm: SHM::SEND
2  Input: Send buffer b, destination d
3  increment ref count for b
4  if d has a matching receive posted then
5  |  if the posted receive is receive_copy then
6  |  |  signal receive condition variable with b
7  |  else
8  |  |  release(receive buffer)
9  |  |  point receive buffer to b
10 |  |  signal receive condition variable with DONE
11 else
12 |  add b to receive queue of d as send
```

```
13 Algorithm: SHM::SEND_COPY
14 Input: Buffer b to be sent, destination d
15 if d has a matching receive posted then
16 |  copy b to receive buffer
17 |  signal receive condition variable with DONE
18 else if b is not in shared memory then
19 |  n = shm::alloc(sizeof(b))
20 |  copy b to n
21 |  add n to receive queue of d as send
22 else
23 |  increment ref-count for b add b to receive queue of d as
   |  send_copy
```

```
24 Algorithm: SHM::RECEIVE
25 Input: Receive buffer b, source s
26 if there is a matching send from s in the queue then
27 |  if send is shm::send then
28 |  |  release(b)
29 |  |  point b to send buffer
30 |  else
31 |  |  if send buffer's ref-count is 1 then
32 |  |  |  point b to send buffer
33 |  |  else
34 |  |  |  copy send buffer to b
35 |  |  |  release(send buffer)
36 else
37 |  wait for send as receive
```

```
38 Algorithm: SHM::RECEIVE_COPY
39 Input: Receive buffer b, source s
40 if there is a matching send from s in the queue then
41 |  if send buffer's ref-count is 1 then
42 |  |  point b to send buffer
43 |  else
44 |  |  copy send buffer to b
45 |  |  release(send buffer)
46 else
47 |  wait for send as receive_copy
```

Algorithm 1: Outline of `send`, `receive`.

existed before) due to sharing of communication buffers. Section 6 presents source-level compiler transformations for C++ programs that automatically convert a program using MPI to our system. In principle, it is possible to use the run-time system without the accompanying compiler, as long as the user is willing to perform the translation manually.

The run-time system consists of an application programming interface (API) and a backend that controls access to the shared memory. The API has a function corresponding to each MPI communication primitive. Figure 2 diagrams the system. The box labeled "book keeping" contains certain global data structures, including a receive queue for each processor. Sender and receiver synchronize on the receiver queue and messages are checked for receiver rank, tag, and communicator for a match, as in MPI.

The call to `MPI::Init` is replaced by `shm::init`, which allocates shared memory, attaches a pointer to it, and forks the required number of processes. All MPI constants are recognized and honored by the API. After the processes have been created, each has the shared memory region available in its virtual address space. Since the processes are forked after a pointer has been attached, each process sees identical virtual addresses for the shared memory region.

A call to `shm::alloc` returns a buffer, as shown in the figure. However, the pointer returned to the application is at a slight offset from the beginning of the buffer, to accommodate two fields: a reference counter, and a condition variable to control concurrent access by multiple processes. Appropriate padding is used to align the returned pointer to cache block boundaries.

Algorithm 1 outlines the algorithms used by various `send` and `receive` primitives. The handling of wildcard in receive (RECEIVE_ANY) has been omitted from the outlines, for the sake of clarity. Other primitives are implemented similarly.

A key difference between MPI and our run-time system is in the signature of `receive` and other functions that accept a receive buffer. `shm::receive` expects the receive buffer to be passed as a mutable reference. The compiler ensures that all communication buffers passed to `shm::receive` are mutable references, as described in Section 6. If a matching `shm::send` has already been posted by the time the receiver calls `shm::receive`, the run-time simply changes the receive pointer to point to the send buffer and signals the receiver condition variable to indicate that `shm::receive` has finished. If no matching `shm::send` has yet been posted, the process calling `shm::receive` waits on the condition variable associated with the buffer. `shm::send` works analogously, as shown in Algorithm 1.

In addition to the shared memory, the runtime system maintains a per-process receive queue (not shown in the figure). When a process calls `shm::send`, the run-time system checks the destination process's queue for any outstanding `shm::receive` call that may be satisfied by it. In that case, the buffer pointer is copied to the receiver and a "wake-up" signal sent to the receiving process through a condition variable. Otherwise, the send buffer is added to the destination queue. Reader count associated with each buffer helps the run time system coordinate concurrent accesses and buffer deallocation, as discussed in Section 3.

In certain cases it may not be possible to determine the exact size of a communication buffer or the communicated buffer may be part of a larger aggregate data structure. In these cases, it might be necessary, or worthwhile, copying the data.

The run-time API provides `*_copy` version of each communication primitive that can accept a process-local buffer address. These versions also accept shared memory buffers, with the semantics that the data still need to be copied from the sender to receiver. This is useful, because when communication involves buffers from shared memory then one buffer copy can be avoided, even when copy semantics are

```
1  Algorithm: SHM::ALLOC
2  Input: buffer size sz in bytes
3  n = allocate sz bytes on shared memory
4  set ref-count to 1 for n
5  return n
---
6  Algorithm: SHM::RELEASE
7  Input: buffer b
8  decrement ref-count for b
9  if ref-count is 0 then
10 |deallocate b from shared memory
---
11 Algorithm: SHM::CHECK_CLONE
12 Input: buffer b
13 lock on b's mutex
14 count = get ref-count for b
15 if count is 1 then
16 |n = b
17 else
18 |n = shm::alloc(sizeof(b))
19 |copy b to n
20 |decrement ref-count for b
21 unlock on b's mutex
22 return n
```

Algorithm 2: Outline of `alloc`, `release` and `check_clone`.

involved. One scenario where the copy-version of a primitive might be invoked on a shared memory address is when the communication primitive is called inside a function on one of its arguments and different call sites might pass different types of buffers to it.

COROLLARY 1. *Accesses to shared buffers are race free.*

PROOF. Writes happen only on buffers with ref-count one. This follows from the placement of `check_clone` calls before writes to the buffer. If the ref-count is one, the buffer is reused. If not, it is copied and the new ref-count is set to one, and the original count is decremented signifying that the caller has released the buffer. □

COROLLARY 2. *The shm runtime does not leak memory.*

PROOF. Note that the ref-count eventually reaches zero and the last release frees that buffer. By Theorem 1, one and only one release will be placed on each path from the shm call to function end (post-dominance by a set). The only operations that decrease the ref-count are release and check_clone (Algorithm 2). The copy operations decrement the counter (through `release`) and copy if and only if the ref-count is greater than one. If ref-count is one, the buffer is reused. An important detail is that send (not `send_copy`) increments the ref-count. All of this implies that the transformations introduce no new memory leaks. □

COROLLARY 3. *The translation preserves semantics.*

PROOF. The only parts of the source code we change are buffer allocations (only communication buffers are changed), and MPI calls. Scopes are preserved and no dependencies are violated (per observations 1, 2). Synchronization on the receiver queue, as mentioned above, ensures that sends and receives work correctly. □

To further optimize performance we also implemented a rendezvous scheme in the runtime for point-to-point communication. Since this guarantees both send and receive buffers to be available, it often eliminates one buffer copy. When copying data between two buffers that are both in shared memory, senders and receivers share the copying work, which almost doubles the effective memory bandwidth. Finally, when copying data between two local buffers (i.e., necessitating two copies), the process is pipelined between the sender and the receiver. A similar mechanism is used by OpenMPI on shared memory for pipelining buffer copying. Rendezvous must be enabled or disabled for the entire application. Since we do not rely on statically matched sends and receives it is not possible to mix the two versions.

6. COMPILER TRANSFORMATIONS

Discussion in Sections 4 and 5 leads directly to compiler algorithms for transforming MPI code. The transformations consist of:

1. Replacing calls to MPI primitives by equivalent calls to our library functions.
2. Changing allocation and deallocation of communication buffers.
3. Inserting calls to resolve write conflicts in order to preserve the original semantics of the program.

Note that we do not require the compiler to be aware of the MPI semantics in depth. Thus, we do not need to match MPI send and receive operations, which could be complicated and the consequent inaccuracies in that analysis would negatively impact performance [3]. We do, however, assume that the original MPI program is correct in that it does not deadlock due to mismatched sends and receives. Similarly, we do not need to assume any fixed number of processes, which is frequently done to make static analyses possible in the presence of MPI calls [7].

Algorithm 3 outlines the overall algorithm, omitting the handling of aliases to avoid cluttering the primary discussion. The remainder of this section describes each of the three major steps in the algorithm.

6.1 Replacing MPI Function Calls

All MPI functions of the form `MPI::`p can be safely replaced by `shm::`p`_copy` using our library, which is what the compiler starts by doing. It next verifies the conditions of Theorems 1 and 2 for each communication buffer to progressively replace each send or receive primitive. For those primitives that involve both sending and receiving (such as `sendrecv` or `alltoall`) the conditions are checked separately for sending and receiving. In some cases when the send and receive buffers are common, such as in `bcast` or when in-place option has been specified (discussed later in Section 7), the buffer must satisfy the conditions for both send and receive for the operation to become the no-copy version.

The conditions are embodied in `NoCopyNeeded` that returns the boolean value `true` only if a given communication function, `shm::`p`_copy`, may be safely replaced by `shm::`p. This step, Step 1b, is actually performed after Step 2 for efficiency. After Step 2, we know exactly which allocations were successfully moved to shared memory.

6.2 Memory Management

Algorithm 4 outlines the steps needed to gather all the

```
1  Algorithm: MPI-to-SHM
2  Input: Function, F, to be transformed
3  Output: Modified function F
   ---------------------------------------------------------
   /* Step 1:  Rewrite MPI calls                          */
4  for each MPI primitive MPI::p in F do
5      Replace MPI::p by shm::p_copy

   /* Step 2:  Change buffer allocation                   */
6  C ← all buffers involved in shm calls in F
7  for each communication buffer c in C, for primitive
   shm::p_copy do
8      d ← data object within which c points
9      if d ≠ c then
10         next
11     alloc ← GetAllocation(F, c)
12     if alloc.type == STACK or STATIC then
13         Replace alloc by shm::alloc(c) at the beginning of
           c's scope
14         Add shm::release(c) at the end of c's scope
15     else if allocation == HEAP then
16         Replace alloc by shm::alloc(c)
17         Replace de-allocation of c by shm::release(c)
18     else
           /* Allocation is unknown, do nothing           */

       /* Step 1b:  Convert to non-copy versions,
          wherever possible                                */
19     if NoCopyNeeded(c) then
20         replace shm::p_copy by shm::p

   /* Step 3:  Add cloning checks                         */
21 C ← all buffers involved in shm calls not ending in _copy
22 for each communication buffer c in C do
23     for each write statement, w, into c that is reachable from
       a communication step involving d do
24         if w is in a loop then
25             insert check_clone(c) before the loop-nest
               containing w, but not containing any reaching
               communication statement
26         else
27             insert check_clone(c) just before w
28 return F
```

Algorithm 3: Simplified overall algorithm to transform an MPI program, assuming no aliasing.

allocations that need rewriting, inter-procedurally, but ignoring aliasing for the sake of clarity. The algorithm traverses the def-chain (inter-procedurally, if required) until it finds an allocation. All reaching allocations are collected in set H or S, depending on whether the allocation was heap- or stack-based (stack-allocation set also includes static or global allocations). Subsequently, the sets can be used to rewrite allocations and deallocations of each buffer. The algorithm is called once for each function with B set to the empty set, ϕ.

Note that the algorithm "gives up" when it cannot trace the definition point to a known type of statement. For example, if the buffer was passed by reference to a function or assigned a return value from some unknown function. In such cases, the buffer simply does not appear in either set H or set S, reflecting the fact that buffer has an unknown allocation and its allocation remains unchanged. Also note that the function `NoCopyNeeded` may be implemented using the

```
1  Algorithm: GatherAllocations
2  Input: Function F, starting buffer B
3  Output: Stack-allocated buffers S, heap-allocated buffers
           H
   ---------------------------------------------------------
4  H ← φ
5  S ← φ
6  if B = φ then
7      B ← communication buffers in F
8  for each communication buffer b in B do
9      z ← size-expression in communication primitive using b
10     if z < threshold then
11         continue
12     Q ← getDefs(b)
13     while not Q.empty do
14         d ← Q.extract
15         if d is malloc or new and the size matches z then
16             H ← H ∪ {d}
17         else if d is a declaration and the size matches z then
18             S ← S ∪ {d}
19         else if d is a copy constructor or simple assignment
           then
20             Q.add (right hand side of d)
21         else if d is a formal parameter then
22             for each caller C of F do
23                 for each call-site in C that calls F do
24                     a ← actual parameter corresponding to d
25                     (h, s) ← GatherAllocations(C, {a})
26                     H ← H ∪ h
27                     S ← S ∪ s
28 return (H, S)
```

Algorithm 4: Algorithm to gather all stack- and heap-allocated communication buffers that can be allocated in shared memory, ignoring aliasing for the sake of clarity.

sets H and S.

The compiler inserts a `shm::release` call when the buffer is no longer live. There are three cases:

1. If the buffer was a stack-allocated object in the original program then its scope is statically determined and the end of its original scope is a natural point where the buffer can be released.
2. If the buffer was dynamically allocated and the original program contains a deallocation statement for the buffer then the deallocation is converted into a release.
3. If the buffer is overwritten before it is freed or its scope ends then the compiler inserts a special `check_clone` call before the code that modifies the buffer. That call implicitly releases the old buffer if the buffer is being shared by more than one process and it is cloned (see Section 5). As an optimization, if the compiler can prove that the code that modifies the buffer completely overwrites the old values then it inserts a call to `check_alloc`, which does not copy the old values. Similarly, `shm::receive` and `shm::send` may also implicitly release the old buffer.

In addition, if the compiler finds no reuse of a send buffer, `send_release` is called to immediately release the send buffer, instead of `send`. This special case is also called *ownership transfer* [13] or *ownership passing* [6].

The compiler tried to insert the `check_clone` call closest to the statement that modifies the shared buffer, so that we

give the maximum possible time to any consumers of that buffer to avoid cloning or allocating a new one. However, if the use is in a loop the `check_clone` is inserted before it to avoid repeated redundant checking.

7. COMPLICATIONS

The factors that complicate automatic translation of MPI-based code to that using shared-memory fall into two broad categories, those caused by the language—C++ in our case—and those due to the MPI library. It is important to remember that in each case, the compiler always has the option to fall back to `shm::*_copy` primitives, which are guaranteed to preserve the original semantics, by Theorem 1.

7.1 Language-related Issues

Automatically allocated buffers.

A static or stack-allocated variable declared as "`T buf`", of type `T` is converted to `typedef T newT; newT* buf = shm::allocate(sizeof(T))`. Using `typedef` allows the compiler to handle complicated declarations involving array subscripts and expressions.

Pointers.

Standard "points-to" data flow analysis, to determine which variables a pointer may point to, helps us track the origins of a communication buffer. If static points-to analysis fails, the compiler falls back to generating `copy` version of the corresponding communication primitive.

Multiple reaching allocations.

With pointers, a communication buffer could be allocated in different ways along different paths reaching the use a communication buffer. This could happen inter-procedurally when the buffer is passed as an argument into the function that uses it and allocated differently in different callers.

```
double *p;
if (some_condition)
  p = allocate_one_way();
else
  p = allocation_another_way();
...
MPI_send(p, ...)
```

Figure 3: A communication buffer may be allocated differently along different control-flow paths.

In such cases, the compiler can safely convert `MPI::Send` to `shm::send` only if all reaching allocations of the buffer satisfy the conditions necessary for shared-memory allocation. Similarly, if a buffer could be an alias of other buffers then each of the aliased buffers must satisfy the necessary conditions before the compiler can make the change.

Aliasing.

One way to handle aliases is to replace the original buffer variables, and their aliases, by references. This causes an extra level of indirection, but whenever a buffer is updated to point to a new location, all aliases automatically get updated. In the code examples we studied we did not have to use this extra level of indirection in any case.

Size of shared memory.

Modern operating systems do not allocate any physical memory corresponding to a virtual page, until it is actually used [15]. We leverage this feature by allocating a large pool of virtual shared memory in the beginning to which each process attaches. We have found the extra initial kernel-level cost of populating the page table to be negligible, in practice.

An alternative strategy is to expand the shared memory as needed. However, this will require considerably more bookkeeping and might require the processes to be synchronized and made aware of the increased amount of shared memory, when it is expanded. Our solution is simpler as well as relatively inexpensive.

7.2 MPI-related Issues

Matching sends and receives.

Matching sends with receives in a program pose a difficult challenge in statically analyzing MPI programs [3]. Our approach side-steps this issue by relying on a specialized runtime system that matches sends with receives at run time, just as an MPI library would. Instead, the performance benefit is derived from unifying send and receive buffers at run time.

Wildcard receive.

MPI library allows a process to post a receive request to receive from any potential sender, without specifying the sender. Since our approach does not rely on statically matching sends with receives, wild card receives do not pose any additional problem—the receiver buffer starts pointing to whichever send it matches at run time.

Collective operations.

MPI collective operations involve multiple processes. That by itself does not pose any complication. However, while some of these collectives communicate entire local buffers to remote processes, such as, `MPI::Bcast` (broadcast) and `MPI::Reduce`, others communicate only parts of a larger buffer to each remote process, such as `MPI::alltoall` and `MPI::allgather`. This issue is discussed in Section 7.3.

Non-blocking communication.

MPI supports non-blocking (asynchronous) versions of most communication primitives, including collectives. The copy versions (`shm::*_copy`) of the non-blocking primitives use a separate thread to perform the data copying. The copying thread is created in advance and waits on a semaphore to get work from the main thread whenever it is not actively copying data. The lightweight 0-copy versions are converted to their blocking variants.

MPI communicators.

Communicators provide a mechanism for MPI processes to form groups and execute collective operations locally within those groups or across groups. Since communicators do not change the mechanism of data communication, they do not pose any additional technical hurdles in optimizing communication with shared memory. We currently support a subset of communicators-based primitives.

In-place option.

MPI primitives that accept two separate buffers for sending and receiving, such as `MPI::Sendrecv`, support an option to reuse a single buffer for sending and receiving, known as the *in-place* option. This means that the run time system must copy the send data into an internal buffer to avoid overwriting it. This internal buffer is allocated in shared memory and necessitates one copy.

One-sided communication.

Recent MPI standards also support one-sided communication, but these are currently far less popular than matched send-receives and collectives. Our current solution does not handle one-sided primitives, although, it is possible to apply our ideas to move one-sided communication buffers into shared memory.

MPI data types.

MPI supports creating runtime data types, that can be used to send a receive non-contiguous chunks of data with regular patterns. Our run time implementation does not support that at this time, although there is no technical issue in supporting it. The compiler assumes these to be opaque types and resorts to using `copy` versions of the communication primitives.

7.3 Other Issues

Sending partial buffers.

It is not uncommon for programs to directly communicate portions of a larger data structure, for example, a single row or column of a 2D array. However, the much higher cost in cloning could obliterate any gains made by avoiding copying while sending. If the compiler can prove that the sender buffer is not overwritten before a global synchronization (such as a barrier), the buffer may be safely sent in-place. Fields of a `struct` or members of a `class` can be handled similarly.

Overlap regions.

A special case of communicating parts of larger data structures is overlap (also called shadow or halo) regions. As a programmatic idiom, shared areas, called the overlap regions, of a distributed array are often duplicated across the processors that would need those. Further research will be needed to evaluate the benefits of automatically identifying and eliminating overlap regions.

Non-uniform memory access.

Simply changing the receiver's pointer to the send buffer may not result in the desired performance benefit when memory-access is non-uniform. For instance, if the send buffer has a higher access cost for the receiver then it might be worthwhile copying the data into an area that is closer to the receiver. Thus, heterogeneous access latencies will make the decision of buffer allocation and copying significantly more complex.

8. EVALUATION

We have implemented our run-time system, using the Boost Interprocess library [2]. It supports all major MPI primitives. The compiler transformations have been implemented using the ROSE source-level compiler infrastructure for C and C++ [14]. The compiler is able to automatically translate all MPI primitives to `shm` calls and change allocations inter-procedurally. The experiments reported here used the compiler to translate the code.

We evaluated our runtime and compiler system on a set of micro-benchmarks designed to compare the performance of our shared memory primitives compared to MPI tuned for shared memory. We also evaluated our system on eight benchmarks, including two dense matrix kernels, Cholesky and Jacobi, an N-Body simulation code, and five NAS parallel benchmarks, DT, IS, MG, LU, and CG [1]. We selected cyclically column-distributed Cholesky for an example of dense matrix computation with complex communication patterns. We implemented a two-dimensional block-distributed Jacobi to include a mix of contiguous (row) and non-contiguous (column) buffer communication. The NAS benchmarks include those with sparse data and irregular communication patterns, scenarios that are often difficult or impossible to handle in traditional MPI-aware compiler analysis.

All our experiments were conducted on the NSF-funded FutureGrid infrastructure. We used a machine based on Intel Xeon 5660 CPU running at 2.80 GHz, with 2 CPUs and 12 cores per node. Thus, we did not exceed 12 processes in any of our experiments to match the number of cores on a single node, except N-Body in which the number of processes is determined by the number of particles. All code was compiled using gcc 4.1.2 on Linux 2.6 and compared with OpenMPI 1.4.3 optimized for shared memory.

8.1 Micro-benchmarks

Figure 4 shows the comparison of seven MPI primitives with their equivalent `shm` versions. There are four lines for `shm`, one for zero-copy version, two for single-copy versions (receiver- or sender-side copy), and one for two-copies version. Each graph plots the time spent in the primitive with exponentially varying buffer size (powers of 2) on the horizontal axis. All collectives were tested with 12 processes. Lines marked with "1-copy (send)" indicate the case when `send_copy` was matched with `receive`. Similarly, "1-copy (recv)" indicates `send` matched with `receive_copy`. "2-copies" indicates both primitives using the `_copy` version. For broadcast, sender is the root and all others are receivers.

The performance of `alltoallv` (not shown) is almost identical to `alltoall`. MPI performance is highly variable for small buffer sizes, especially for `scatter` and `gather`.

As expected, the overhead for non-copy versions remains nearly flat with buffer size for most primitives. However, even for primitives that require going over the buffer, such as `reduce`, our library beats the performance of the well-tuned OpenMPI library.

When `_copy` versions are used, the performance is closer to MPI, which is not unexpected. Only `sendrecv`, with two copies, performs worse than MPI. We believe that the highly tuned memory copying in MPI is able to beat our simple memory copy in this case. However, in practice, a program might contain a mix of copy and non-copy versions of primitives. Therefore, the amortized cost of a communication primitive is likely to lie somewhere between the extremes represented by the lines for `shm` library.

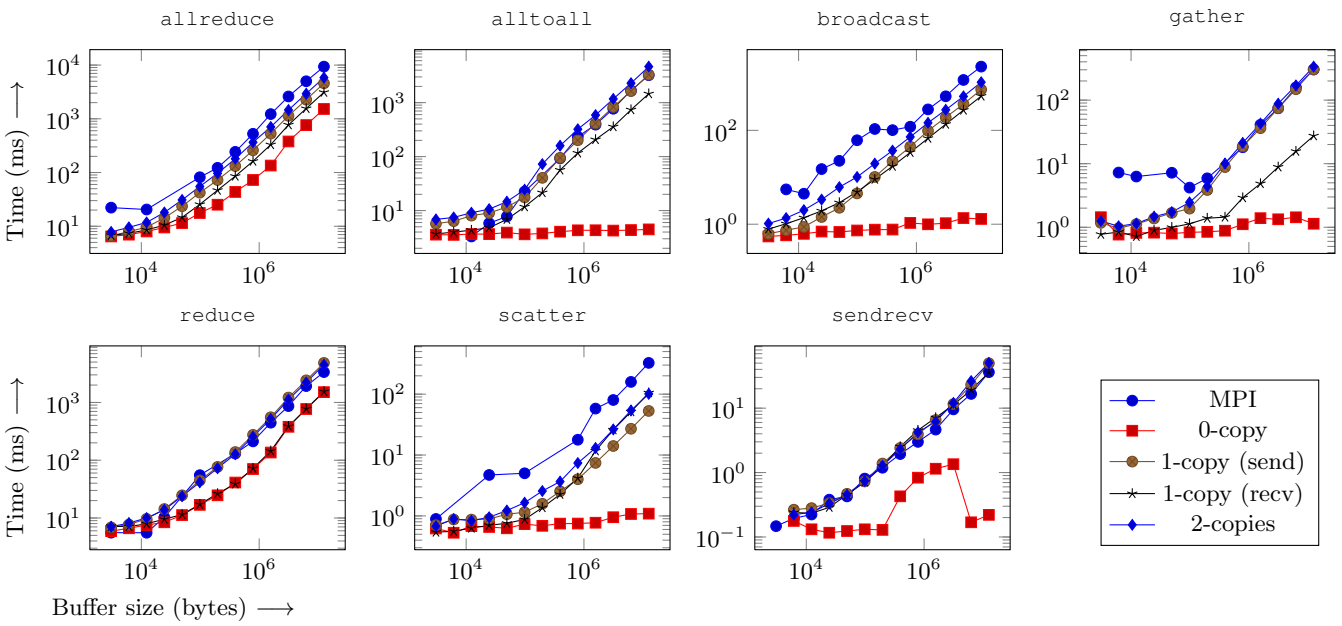

Figure 4: Comparing seven MPI primitives with their `shm` counterparts with 12 processes.

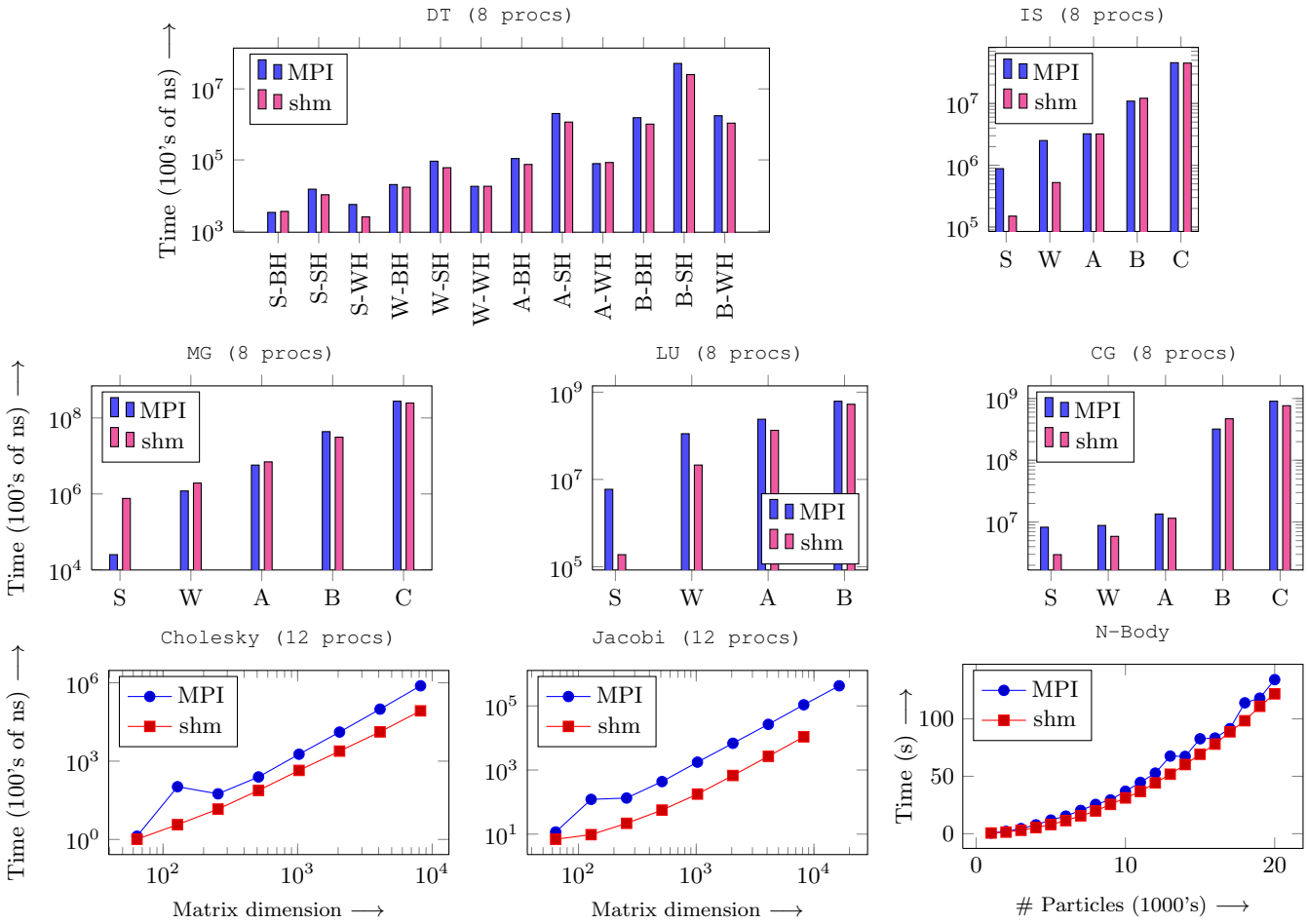

Figure 5: Shared-memory performance of translated code compared to original MPI.

8.2 Application Benchmarks

We evaluated the running times for a variety of input sizes for Jacobi and Cholesky. NAS benchmarks were run for all the supported classes, except CG that failed to run to completion for class C on our testbed. Figure 5 summarizes the range of speedups observed over MPI. The number of processes for N-Body is determined by the number of particles and generally far exceeds the number of cores. Note that the vertical axis is logarithmic for all graphs, except N-Body.

All NAS benchmarks were tested using 2, 4, and 8 processes, but results for only 8 processors are shown, as others are similar. We note that the shared-memory version almost always performs better or no worse than MPI. The only exceptions are IS and CG for classes B, and MG for class A. We implemented a rendezvous protocol for data copying to be able to parallelize it between the participating processes. However, sometimes that can introduce unnecessary waiting.

DT is consistently improved, but IS shows less improvement for large input sizes and CG actually runs significantly slower. Most NAS benchmarks use asynchronous communication. As a result they are able to effectively hide latencies. However, shared-memory versions perform equally well without having to start with asynchronous MPI calls, which eliminates the need to write hard-to-maintain code. Cholesky has a broadcast in a loop that gets translated to 0-copy, which significantly improves performance. In Jacobi, the original code uses non-blocking receives to hide latencies. Nevertheless, using shared memory significantly improves performance.

9. RELATED WORK

Several past efforts have tried to optimize MPI on shared memory.

Thread based approaches.

Tang et al. [16] and Friedley et al. [6] have separately developed compiler transformations for converting MPI to threaded programs. While threaded code has certain advantages, it has certain inherent limitations, for example, in not being able to work with thread-unsafe libraries, closed-source libraries that use global states, and MPI programs that are multi-threaded to start with.

Ownership passing.

Friedley et al. [6] also presented a static analysis to convert legacy MPI code into code that uses their ownership passing library built on top of MPI. Their approach is of "eager copying" of communication buffers, against ours of "lazy copying." For example, in the code in Figure 6 ownership passing will create $N-1$ copies where N is the total number of processes. Our approach will convert `Bcast` to a zero-copy version, eliminating all copying.

```
1 Data *buffer;
2 while (1) {
3   buffer = new ...;
4   buffer = ...
5   ...
6   MPI::Bcast(buffer, ...);
7   ... = buffer;
8   delete buffer;
9 }
```

Figure 6: Example code to illustrate the difference between ownership passing and our approach.

Also, our approach does not need the non-trivial send-receive matching required for their analysis. It is not clear from their paper what happens when information about sender or root is not available to the analysis (e.g., buffer could have been packed in a different method).

Negara et al. [13] and Friedley et al. [6] discuss ownership transfer, or ownership passing, for efficient message passing. Their analyses are based on the actor model of communication. This is a special case of our approach when sender transfers its reference count to the receiver, by calling `shm::send_release`. Our system encourages buffer reuse when concurrent writes do not occur, leading to potentially better cache performance.

Other.

Buntinas et al. use a different interface to send large messages and reduce buffering overhead at runtime [4]. Our combination approach of static analysis complemented by a runtime system gives us the best of both worlds—reduced overheads with static analysis, and run-time optimization whenever possible. Our approach is also amenable to a post-conversion manual optimization pass.

Jiao et al. [10] proposed a copy-on-contention approach for optimizing message passing on shared memory, but they start with a domain-specific language for communication, which gives them much better control over analysis and code generation.

MPI-aware compilation to overlap computation and communication has been implemented at various times in the past [11, 5]. These strategies are aimed at narrow objectives requiring interpretation of only certain characteristics of the MPI library.

Some researchers have proposed type systems to infer ownership of objects and proposed algorithms to infer them automatically [8]. However, these type systems remain too restrictive.

10. CONCLUSION

We have presented an approach to optimize communication in MPI programs on shared memory. Our approach is based on a combination of a run-time system that replaces the standard MPI primitives, and a source-level compiler framework that automatically translates MPI programs to use it. The run-time system uses a *copy-on-conflict* strategy to clone buffers only when a conflict arises due to multiple concurrent writers. This encourages buffer reuse, leading to smaller memory footprint and better locality properties than using write-once buffers exclusively, while still providing the benefit of zero-copy communication across processes that share physical memory. Our compiler algorithms build on standard compiler analyses to place zero-copy communication, whenever it is provably safe to do so, and defer the copy-optimization to the run-time system for the remaining cases. This hybrid strategy avoids the difficult problem of statically matching sends and receives, making it more broadly applicable and provides the well-known *ownership passing* for free whenever applicable. Moreover, maintaining address space separation makes our technique applicable even in the presence of legacy thread-unsafe libraries.

11. ACKNOWLEDGEMENTS

We thank Greg Bronevetsky and Andrew Friedley for initial discussions and Jeremiah Willcock for providing feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0834722.

12. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. DOI: 10.1177/109434209100500306.

[2] Boost C++ libraries. `http://www.boost.org/`.

[3] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 1–12. IEEE Computer Society, 2009. DOI: 10.1109/CGO.2009.32.

[4] D. Buntinas, G. Mercier, and W. D. Gropp. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *PVM/MPI'06*, pages 86–95, 2006.

[5] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 316–325, 2009. DOI: 10.1145/1542275.1542321.

[6] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: Efficient distributed memory programming on multi-core systems. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 177–186, 2013. DOI: 10.1145/2442516.2442534.

[7] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. D. Supinski, M. Schulz, and G. Bronevetsky. Formal analysis of MPI-based parallel programs. *Communications of the ACM*, 54(12):82–91, Dec. 2011. DOI: 10.1145/2043174.2043194.

[8] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978. DOI: 10.1145/359576.359585.

[10] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011. DOI: 10.1109/HiPC.2011.6152733.

[11] K. Kennedy and A. Sethi. Resource-based communication placement analysis. In *Proceedings of the 9th Workshop on Language and Compilers for Parallel Computing (LCPC)*, pages 369–388. Springer-Verlag, 1996.

[12] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard, Version 2.2*, Sept. 2009.

[13] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 81–90, 2011. DOI: 10.1145/1941553.1941566.

[14] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi. *ROSE User Manual: A Tool for Building Source-to-Source Translators*. Lawrence Livermore National Laboratory, Livermore, CA 94550, version 0.9.5a edition, Nov. 2010.

[15] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 8th edition, 2008.

[16] H. Tang, K. Shen, and T. Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):673–700, July 2000. DOI: 10.1145/363911.363920.